mck8s: An orchestration platform for geo-distributed multi-cluster environments

Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson, Erik Elmroth

mck8s: An orchestration platform for geo-distributed multi-cluster environments

Mulugeta Ayalew Tamiru
Elastisys AB
Univ Rennes, Inria, CNRS, IRISA
mulugeta.tamiru@elastisys.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
guillaume.pierre@irisa.fr

Johan Tordsson, Erik Elmroth
Elastisys AB
{tordsson,elmroth}@elastisys.com

***Abstract*—Following the adoption of cloud computing, the proliferation of cloud data centers in multiple regions, and the emergence of computing paradigms such as fog computing, there is a need for integrated and efficient management of geo-distributed clusters. Geo-distributed deployments suffer from resource fragmentation, as the resources in certain locations are over-allocated while others are under-utilized. Orchestration platforms such as Kubernetes and Kubernetes Federation offer the conceptual models and building blocks that can be used to build integrated solutions that address the resource fragmentation challenge. In this work, we propose mck8s – an orchestration platform for multi-cluster applications on multiple geo-distributed Kubernetes clusters. It offers controllers that automatically place, scale, and burst multi-cluster applications across multiple geo-distributed Kubernetes clusters. mck8s allocates the requested resources to all incoming applications while making efficient use of resources. We designed mck8s to be easy to use by development and operation teams by adopting Kubernetes' design principles and manifest files. We evaluated mck8s in a geo-distributed experimental testbed in Grid'5000. Our results show that mck8s balances the resource allocation across multiple clusters and reduces the fraction of pending pods to 6% as opposed to 65% in the case of Kubernetes Federation for the same workload.**

***Index Terms*—Resource provisioning, placement, scheduling, autoscaling, Kubernetes, multi-cluster**

I. Introduction

Virtualized computing infrastructures are increasingly geo-distributed. For reasons such as high availability, low user-perceived latency, privacy, and compliance with national regulations, many infrastructures are being designed as a set of server clusters located in different regions [1].

Managing large-scale applications in these environments is a difficult challenge. Geographical resource distribution increases fragmentation where the resources in one location may be overloaded while those in another location may remain under-utilized. It is therefore important to provide users with simple yet powerful ways to control the scale and location of their replicated applications. When an application running in its preferred location runs out of resources, it may need to acquire additional resources in the local cluster or, if the local cluster runs out of resources, in another nearby cluster. In extreme cases where no suitable cluster resources may be found, the platform may need to burst to a public cloud where additional resources may be rented for the duration of the overload, then decommissioned when the workload decreases.

We base this work on the popular Kubernetes (*k8s*) container orchestration platform. *k8s* has fully demonstrated its ability to efficiently orchestrate the resources within a single cluster. However, it does not implement any notion of resource location and therefore does not allow its users to control the location of resources assigned to run their applications [2]. Kubernetes Federations (KubeFed) extend *k8s* with an explicit notion of multi-cluster environment. However, in its present form, KubeFed's main focus is on the manual placement of resources on selected clusters and implements only one generic automated scheduling mechanism. The lack of automation limits KubeFed's ability to manage a large number of clusters, whereas the absence of policy-based scheduling prevents users of multi-cluster deployments from specifying their desired scheduling preferences, and limits the efficient use of resources.

To address these challenges, we propose mck8s, a general-purpose integrated orchestration platform for multi-cluster computing environments that offers multi-cluster scheduling with various placement policies, multi-cluster horizontal pod autoscaling, and dynamic cloud cluster provisioning capabilities to automate the deployment, resource provisioning, and scaling of multi-cluster applications. Our platform is built as an extension of *k8s*, KubeFed, and other leading cloud-native tools. Our work has the following objectives: (1) Maximize resource utilization in multi-cluster environments; (2) Guarantee that all applications submitted to a multi-cluster environment find a place to run by making use of all existing resources and provisioning additional resources from the cloud; (3) Maintain the performance of applications by adjusting the number of replicas in response to changing user traffic; and (4) Guarantee that resources are not over-provisioned and wasted unnecessarily when they are not being used. Our experimental results (Section VI) in a geo-distributed multi-

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

cluster environment complemented with resources from the cloud show that mck8s reduces resource fragmentation by balancing resource allocation for multi-cluster applications, and keeping the percentage of pending pods below 6% as compared to 65% in the case of KubeFed for the same workload.

II. Background

KubeFed allows application deployment and resource management on multiple clusters, called member clusters, from a host cluster that hosts the KubeFed control plane. KubeFed builds upon Kubernetes using so-called "Custom Resource Definitions" (CRDs) and introduces new abstractions such as FederatedNamespaces, FederatedDeployments, FederatedServices and FederatedJobs that help to conceptualize multi-cluster applications. In fact, KubeFed allows federating any *k8s* resources to be used in a multi-cluster environment.

In KubeFed, users can create and deploy federated resources using declarative manifest files in the usual *k8s* fashion. A manifest file for a federated resource contains three parts, namely, $template$, $placement$ and $override$. The $template$ defines the aspects of the federated resource that are common across all selected clusters. The $placement$ specifies the clusters selected for hosting the federated resource. The $override$ defines aspects of the federated resource that are specific to certain clusters.

In its current form, KubeFed allows selecting the clusters manually to host the federated resources with limited support for automated policy-based scheduling. Moreover, it simply propagates resources to the target clusters without any prior checks on the availability of resources. Because of this, KubeFed cannot scale to manage hundreds and thousands of clusters that are typical in certain multi-cluster use cases such as fog computing. Furthermore, it fails to manage resources efficiently, causing resource wastage and fragmentation.

To illustrate these problems, we deployed a total of 1,126 federated deployments and jobs replaying the Google cluster traces for one hour on a geo-distributed KubeFed environment containing five *k8s* clusters having different capacities. In the setup, each cluster has one master node and five worker nodes: the nodes of Clusters 1 and 5 have 4 CPU cores and 16 GB RAM, whereas those of Clusters 2–4 have 2 CPU cores and 4 GB of RAM. We distribute deployments and jobs to different clusters according to a binomial distribution. This is reflected in the stacked plot in Figure 1 that shows a wide variation in the number of resources requested from the different clusters. Figure 2 shows the CPU allocation in percent, which is calculated as the ratio of the total CPU request of pods in each cluster to the total cluster CPU. We see that Clusters 2, 3 and 4 are over-allocated up to six, seven, and two times their total CPU capacity, respectively, whereas Clusters 1 and 5 are under-allocated. This is because KubeFed simply deploys the pods on the preferred clusters without checking the availability of resources and does not try to balance the deployment when the preferred clusters run out of resources. As a result, we see that up to 65% of the deployed pods remain in a "pending" state. Moreover, overall across all the five clusters, we see that the CPU allocation reaches up to twice the total CPU capacity offered by the clusters, suggesting the need to provide additional resources.

To address these challenges that have to be solved in order to build a mature orchestration platform for managing resources in multi-cluster environments, we introduce mck8s – a platform that offers not only automated policy-based scheduling but also multi-cluster horizontal pod autoscaling and cloud bursting mechanisms. mck8s builds upon and borrows concepts from Kubernetes and KubeFed. It also integrates other open-source tools such as the Cilium cluster mesh for multi-cluster network discovery and global load balancing, Cluster API for declarative resource provisioning from cloud platforms, Prometheus for monitoring, and Serf for measuring inter-cluster network latencies.

III. Related Work

Ever since the emergence of cloud computing as the dominant computing paradigm, there has been a growing interest in making the best out of the available resources, location, pricing schemes, and offerings of multiple cloud providers. Deploying applications across multiple cloud providers maximizes scale, availability, performance, and fault tolerance. By maximizing choice, it also improves cost efficiency for cloud users and profitability for cloud service providers. Other issues such as avoiding vendor lock-in or complying with regional compliance can also be addressed by using multi-cloud deployments.

Previous works have introduced cloud federation, hybrid cloud, multi-cloud, and aggregated service by brokers to deal with the challenges of interoperability and standardization in different cloud providers [3]. Also, several architectural frameworks and platforms have been introduced to achieve the goal of federated and multi-cloud computing such as RESERVOIR [4], OPTIMIS [5], Contrail [6], to name a few. Other works have looked at optimizing the placement of VMs in multi-cloud environments to optimize performance and cost [7]. In particular, in hybrid cloud scenarios, cloud bursting allows offloading applications from private data center to cloud data centers to handle workload spikes [8]–[14].

In the last few years, we have witnessed the growing adoption of geo-distributed deployments such as hybrid cloud, multi-cloud and fog computing in order to meet non-functional requirements such as proximity, high availability, fault tolerance, and compliance with regional regulations. For instance, fog computing has emerged as a decentralized paradigm that extends cloud computing to where the data is generated and users are located [15], [16]. As resources in a fog computing environment are geographically distributed with heterogeneity in resources and network characteristics and location, the problem of resource management has been revisited by a number of works. Some works focus on optimizing the placement of jobs and services [17], [18], whereas others address the joint problem of placement and autoscaling [2], [19]–[21]. Similarly, in this work, we address the issues of placement and autoscaling together. In contrast to our work, these works

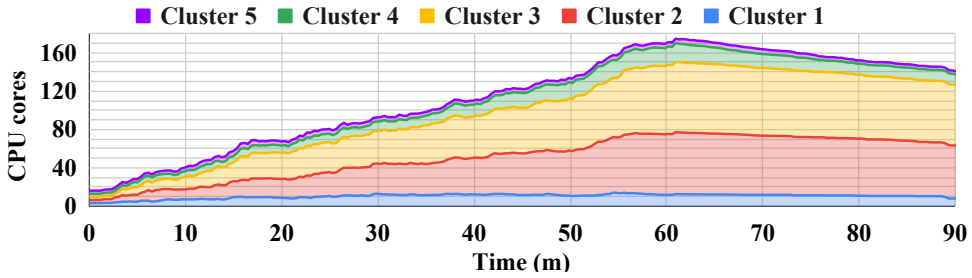


Fig. 1: KubeFed: Total CPU request of pods per cluster.

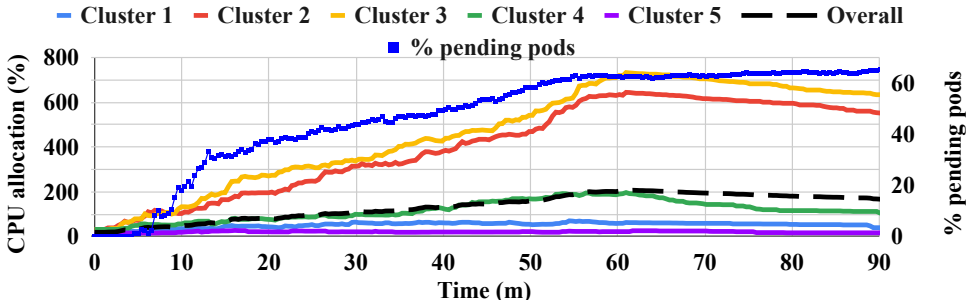


Fig. 2: KubeFed: CPU allocation ratio and percentage of pending pods.

focus on optimizing a single aspect of the placement and propose a single policy, whereas our work is more generic and offers a few placement policies to choose from. Moreover, only some of the works consider optimization at the container as well as the VM level by adjusting the number and size of both containers and VMs [20]. Most of the works assume that there are limited resources available at the fog level. In our work, we also propose a cloud provisioner and autoscaler to augment the fixed resources during times of traffic spike, or when more resources are needed. These works deal with specific aspects of the resource management problem in geo-distributed environments, whereas we propose a generic and integrated orchestration platform to address various aspects of the resource management problem in multi-cluster environments.

Most works to date have focused on conceptual modeling, architectures, and simulation evaluations. As a result, there is a shortage of mature implementations to leverage the benefits that multi-cloud deployments bring to the table. In contrast, in this work, we propose and implement a generic and integrated orchestration platform for managing resources in multi-cluster environments, including multi-cloud deployments, and present results for a real experimental testbed in the hope to fill the gap of lack of implementation and experimental evaluations.

IV. Application Deployment Model

In this paper, we propose mck8s – an integrated container orchestration platform for geo-distributed multi-cluster environments to address the challenges of resource fragmentation. Our platform builds upon *k8s* and KubeFed and offers policy-based multi-cluster scheduling, multi-cluster horizontal pod autoscaling, cloud cluster provisioning and autoscaling, and multi-cluster re-scheduling.

In this section, we introduce the abstractions and Custom Resources (CRs) that make up mck8s. The abstractions and CRs introduced here provide the necessary control mechanisms for creating, updating, and deleting multi-cluster applications. Moreover, different placement policies are proposed to allow flexibility for the orchestration platform to be used under different use cases. Lastly, we have focused on ease of use by making sure that the manifest files used to create these resources as similar as possible to those in *k8s*.

A. Multi-Cluster Deployment (MCD)

A multi-cluster deployment (MCD) consists of a set of deployments on one or more clusters that have the same deployment name. For simplicity, we assume that all the pods of all the deployments in a multi-cluster deployment have the same container image, CPU request, and memory request. However, deployments in different clusters may have a different number of replicas. MCDs hold status information about resource requests, number of replicas, and locations.

Example manifest files of two MCDs are shown in Listings 1 and 2. Similar to KubeFed's FederatedDeployment, mck8s' MCD allows the user to specify the preferred clusters on which the multi-cluster application will be deployed. In contrast, our MCD introduces placement, substitution, and bursting policies to give the user more control, automation, and flexibility in deciding how they want their applications deployed in the multi-cluster environment.

The multi-cluster placement policies are either resource-based (worst-fit and best-fit) or network traffic based (traffic-aware). If substitution is enabled in the case of cluster-affinity placement, substitute clusters are selected if the preferred clusters are incapable of placing the MCD. Similarly, if busting is enabled, an MCD deployed on a single cluster

Listing 1: An example MCD manifest file for traffic-aware placement.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterDeployment
metadata:
  name: mcd-app-1
spec:
  placementPolicy: traffic-aware
  enableBursting: true
  burstingPolicy: nearest-first
  numberOfLocations: 2
  selector:
    matchLabels:
      app: mcd-app-1
      tier: backend
  replicas: 5
  template:
    metadata:
      labels:
        app: mcd-app-1
        tier: backend
    spec:
      containers:
        - name: mcd-app-1
          image: "k8s.gcr.io/hpa-example"
          resources:
            requests:
              memory: 512Mi
              cpu: 500m
            limits:
              memory: 512Mi
              cpu: 500m
          ports:
            - name: http
              containerPort: 80
```

Listing 2: An example MCD manifest file for cluster-affinity placement.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterDeployment
metadata:
  name: mcd-app-2
spec:
  locations: cluster2, cluster5
  enableSubstitution: true
  substitutionPolicy: nearest-first
  enableBursting: true
  burstingPolicy: nearest-first
... (redacted)
```

may be transformed into an MCD on multiple clusters if, for example, its replicas grow in response to user traffic. Unlike the scheduling controller in KubeFed, these policies are integrated into the MCD and are not part of yet another overarching controller, making it easy to use. Moreover, unlike KubeFed, our manifest files are designed to be very much similar to those of vanilla *k8s*, thus allowing existing manifest files for single *k8s* clusters to be easily used on mck8s.

B. Multi-Cluster Job (MCJ)

Similarly, a multi-cluster job (MCJ) consists of a set of *k8s*s jobs that run on one or more clusters. MCJ supports the placement and substitution policies as described in Section IV-A.

Listing 3: An example MCS manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterService
metadata:
  name: mcs-app-1
  annotations:
    io.cilium/global-service: "true"
spec:
  selector:
    app: mcd-app-1
    tier: backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: http
```

C. Multi-Cluster Service (MCS)

The Multi-Cluster Service (MCS) resource manages the lifecycle *k8s* Services corresponding to the deployments under an MCD in the clusters that host them. Unlike KubeFed's FederatedService, it is not required to specify the location(s) for the MCS, it finds the corresponding MCD automatically. Moreover, we present a simpler manifest file similar to that of vanilla *k8s*, as shown in Listing 3. Our MCS is also integrated with Cilium global load balancing to enable routing of user requests to multiple clusters, which is required for bursting.

D. Multi-Cluster Horizontal Pod Autoscaler (MCHPA)

The Multi-Cluster Horizontal Pod Autoscaler (MCHPA) aims to adjust the number of deployment replicas of MCDs in response to changing traffic so that the quality of service provided by the MCDs is maintained. Therefore, it periodically monitors the number of deployment replicas of an MCD in all the clusters they are deployed on, computes the number of desired replicas based on the average resource utilization of the pods of the deployments, and adjusts the number of replicas of the MCD to the desired number of replicas. Unlike *k8s* and KubeFed, MCHPA does not require *k8s*' Horizontal Pod Autoscaler (HPA) to run inside each cluster, rather MCHPA runs inside the management cluster, requiring to define only one resource to manage the scalability of each MCD. As shown in Listing 4, an MCHPA resource can be defined very easily in the same way as an HPA.

E. Cluster Provisioner and Cluster Autoscaler (CPCA)

Whenever additional resources are required to augment the capacity of the fixed clusters, the Cluster Provisioner and Cluster Autoscaler (CPCA) resource interfaces with public cloud services to provision a *k8s* cluster on demand. CPCA is also responsible for adjusting the number of nodes of the cloud clusters and eventually decommission the cloud clusters altogether if not needed for a certain amount of time. The CPCA resource needs to be created only once by using a manifest file show in Listing 5.

F. Multi-Cluster Re-scheduler (MCR)

To make sure that cloud clusters are not overprovisioned, the Multi-Cluster Re-scheduler (MCR) resource periodically

Listing 4: An example MCHPA manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterHorizontalPodAutoscaler
metadata:
  name: mchpa1
spec:
  scaleTargetRef:
    apiVersion: fogguru.eu/v1
    kind: MultiClusterDeployment
    name: mcd-app-1
  minReplicas: 2
  maxReplicas: 100
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 75
```

Listing 5: An example CPCA manifest file.

```
apiVersion: fogguru.eu/v1
kind: CloudProvisioner
metadata:
  name: cp1
spec:
  cloudClusterName: cloud1
  cloudProvider: OpenStack
  floatingIP: 1.2.3.4
  credentials: xxxxxxxxxxxx
```

checks for deployments on the cloud clusters that were deployed because of a shortage of resource on their preferred clusters. The MCR attempts to place these deployments back on the preferred clusters once again. The manifest file needs to be applied only once and requires only the name of the resource as shown in Listing 6.

V. System Design

A. System Model

A multi-cluster environment is defined as a management cluster and a set of n *k8s* clusters $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, where each cluster γ_i has one master node and q_i worker nodes. The architecture of the multi-cluster environment is shown in Figure 3. Even though multiple layers can be supported, for the sake of simplicity, we present a two-layered architecture. At the *management layer*, we find the controllers, tools and cloud clusters, whereas at the *clusters level* we find the member clusters that are controlled by the management cluster.

We assume that each cluster is homogeneous in terms of resource capacity, i.e., each node m_{i_j} in cluster γ_i has $m_i.cpu$ CPU cores and $m_i.memory$ RAM. However, the nodes of

Listing 6: An example MCR manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterRescheduler
metadata:
  name: mcr1
```

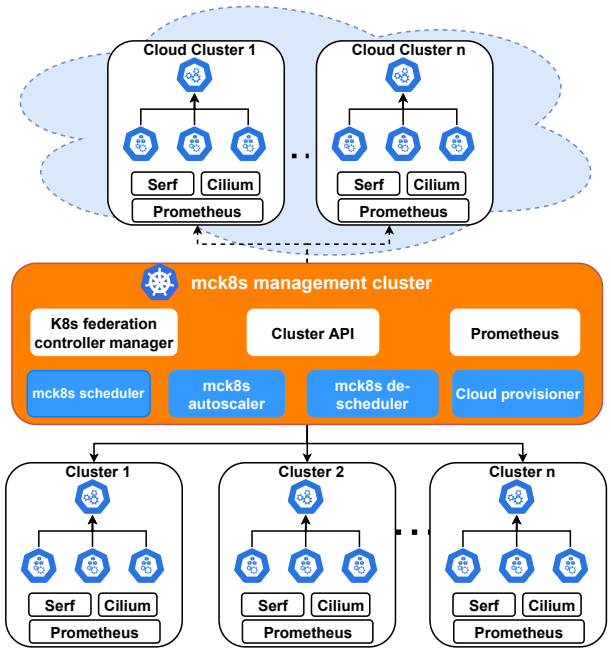


Fig. 3: mck8s architecture.

different clusters may have different capacities. The clusters are geographically distributed, and the inter-cluster latency is defined by the matrix L = $[l_{i_j}]$ as measured by serf.

The management cluster is responsible for monitoring and configuring all member clusters, accepting application deployment requests from users of the system, selecting the right clusters to host the applications, adjust the deployments in response to changes in user traffic, and provision, scale, and deprovision cloud clusters. The member clusters are responsible for accepting application deployment requests from the management cluster and select the nodes that will host the pods of the application and executing them. Moreover, the member clusters are responsible for local monitoring and estimating their distance in terms of network latency from other clusters.

B. Problem Formulation

The multi-cluster scheduling (MCS) problem is to map the deployments of the multi-cluster deployment $\Delta = \{\delta_1, \delta_2, \ldots, \delta_l\}$ to l or more clusters from the set of clusters $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ that have enough available capacity to match the deployment's resource request. The choice of clusters depends on the placement policy specified by the user, the capacity of the cluster nodes, and the resource availability.

The multi-cluster autoscaling problem aims to estimate and allocate at run time the number of deployment replicas per cluster $\delta_i.replicas$ in response to the changing traffic received by the application. This depends on the resource request of the deployments $\delta_i.cpu\_req$ and $\delta_i.mem\_req$ and the target resource usage specified by the user θ_i.

The cloud provisioning/bursting problem addresses resource limitations by dynamically provisioning resources from a

TABLE I: Variables used in system modeling and algorithms.

Variable	Definition
Γ	= $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, set of all clusters
γ_i	$\in \Gamma$, cluster of index i
n	= $\mid \Gamma \mid$, number of clusters
q_i	= $\mid \gamma_i \mid$,number of nodes in cluster γ_i
$\gamma_i.cpu.avail$	Total number of available CPU cores in cluster γ_i
$\gamma_i.memory.avail$	Total amount of available RAM in cluster γ_i
M_i	= $\{m_{i_1}, m_{i_2}, \ldots, m_{i_q}\}$, set of nodes in cluster γ_i
$m_i.cpu$	CPU cores of nodes of cluster γ_i
$m_i.memory$	Memory of nodes of cluster γ_i
B	= $\{\beta_1, \beta_2, \ldots, \beta_m\}$, set of all cloud locations
β_j	$\in$ B, cloud location of index j
Δ	= $\{\delta_1, \delta_2, \ldots, \delta_l\}$, l $\leq$ n, multi-cluster deployment as a set of deployments
$\Delta.clusters$	Clusters on which deployments of Δ are running on
$\delta_i.name$	Name of deployment δ_i
$\delta_i.cpu\_req$	CPU cores allocated to replicas of deployment δ_i
$\delta_i.mem\_req$	Memory allocated to replicas of deployment δ_i
$\delta_i.cpu\_util$	Avg. CPU utilization (%) of replicas of deployment δ_i
$\delta_i.mem\_util$	Avg. memory utilization (%) of replicas of deployment δ_i
$\delta_i.replicas$	Number of replicas of deployment δ_i
$\delta_i.ntk$	Total network traffic received by replicas of deployment δ_i
$P^i_{pending}$	Total number of pending pods in deployment δ_i
$P^i_{running}$	Total number of running pods in deployment δ_i
L	= $[l_{i_j}]$, symmetric n+m x n+m matrix of inter-cluster latencies
$\delta_i.current\_replicas$	Current number of replicas of deployment δ_i
$\delta_i.desired\_replicas$	Desired number of replicas of deployment δ_i
$\delta_i.min\_replicas$	Minimum number of replicas for deployment δ_i
$\delta_i.max\_replicas$	Maximum number of replicas for deployment δ_i
$\Delta.desired\_replicas$	Set of desired replicas for the deployments in Δ
θ_i	Target average CPU utilization (%) for replicas of deployment δ_i

nearby cloud data center such that the ratio of unscheduled (pending) pods is minimized.

The multi-cluster re-scheduler periodically monitors applications currently deployed on cloud clusters for reasons of shortage of resources on the preferred clusters, submits them to the scheduler so that they can be re-scheduled on their preferred clusters. This is also beneficial in terms of cost savings that would result in using the fixed fog results as much as possible and minimize over-provisioning the cloud clusters.

C. Design and Implementation

mck8s addresses the geo-distributed resource management problem using four main controllers that are deployed on the management cluster: *multi-cluster scheduler*, *multi-cluster horizontal pod autoscaler*, *cloud cluster provisioner and autoscaler*, and *multi-cluster re-scheduler*. It builds upon *k8s*, parts of KubeFed, Cluster-API, Cilium, serf and Prometheus. Each controller's design follows the MAPE loop [22] and *k8s* design principles. As a result, the controllers are implemented as *k8s* operators using the Kopf operator framework[1].

[1]https://kopf.readthedocs.io/en/stable/

1) Multi-Cluster Scheduler: The multi-cluster scheduler is responsible for the full lifecycle (i.e., creation, updating, and deletion) of the MCD, MCS, and MCJ resources. When a user submits the specifications for MCD and MCJ resources to the management cluster, the scheduler checks the number of requested resources and placement constraints. Similar to the *k8s* scheduler's approach to select the right nodes to place a pod, mck8s' scheduler goes through two major steps for selecting the right clusters to host the deployments or jobs. In the first step, the scheduler filters out those clusters whose nodes do not have the capacity to place the pods of the deployment. In the second step, the scheduler prioritizes the clusters based on the placement policy specified by the user.

In the case of a "cluster-affinity" policy shown in Algorithm 1, the user specifies a preferred cluster that will have the highest priority if it has already passed the filtering step and has sufficient available resources. Otherwise, a substitution cluster is selected based on the substitutionPolicy specified by the user if the "enableSubstituion" field is set to 'true'.

If preferred clusters are not specified, the placement is done automatically by the scheduler based on the resource-based or traffic-aware placement policies specified. As detailed in Algorithm 2 the filtering step is the same as that of the cluster-affinity case, whereas in the prioritizing step the clusters are prioritized based on the 'worst-fit', 'best-fit' or 'traffic-aware' placement policies. In 'worst-fit', the clusters are sorted in descending order based on the available resources, and the clusters with the largest available resources are selected. On the contrary, with "best-fit" placement policy, the clusters are sorted in an ascending order based on their available resources and the clusters with the least resources are selected. The last policy implemented by mck8s is 'traffic-aware' in which case the clusters are sorted in a descending order based on the amount of network traffic they receive and the clusters that receive the highest amount of network traffic are selected.

If no cluster is found to place the applications, the scheduler updates the status of the resource for the cloud cluster provisioner to be notified and provision a *k8s* cluster in the selected cloud data center.

The scheduler is also responsible for the lifecycle of the MCS resources that manage Service entries for the deployments of the target MCD when a specification for MCS is submitted to the management platform.

2) Multi-Cluster Horizontal Pod Autoscaler: For the applications deployed in the platform to maintain their performance requirements despite changes in traffic, mck8s provides a reactive threshold-based horizontal pod autoscaler. Unlike *k8s* and KubeFed, the controller runs in the management cluster and monitors the resource utilization of the deployment pods. Currently, mck8s supports scaling based on CPU utilization.

The controller manages the MCHPA resource created for each MCD. The controller periodically computes the desired number of replicas based on the target utilization threshold, the current number of replicas, and the average CPU utilization of the pods of the target deployment. If the desired number of replicas is greater than that of the current replicas, the

Algorithm 1: Cluster-affinity placement

Input: Definition for Δ with $\delta.cpu\_req$, $\delta.mem\_req$, $\delta.replicas$ and $\Delta.preferred\_clusters$

for γ in $\Delta.preferred\_clusters$ **do**
  if $\delta.cpu\_req < m.cpu$ and $\delta.memory\_req < m.memory$ **then**
    append γ to $\Delta.eligible\_clusters$
  else
    Get replacement cluster
  if len($\Delta.eligible\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.eligible\_clusters$
    else
      Provision cloud cluster
  else
    for γ in $\Delta.eligible\_clusters$ **do**
      if $\delta.replicas \times \delta.cpu\_req < \gamma.cpu.avail$ and $\delta.replicas \times \delta.memory\_req < \gamma.memory.avail$ **then**
        Append γ to $\Delta.selected\_clusters$
      else
        Get replacement cluster
  if len($\Delta.selected\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.selected\_clusters$
    else
      Provision cloud cluster
Place Δ on $\Delta.selected\_clusters$

Algorithm 2: Policy based placement

Input: Definition for Δ with $\delta.cpu\_req$, $\delta.mem\_req$, $\delta.replicas$ and $\Delta.placement\_policy$

for γ in Γ **do**
  if $\delta.cpu\_req < m.cpu$ and $\delta.memory\_req < m.memory$ **then**
    append γ to $\Delta.eligible\_clusters$
  if len($\Delta.eligible\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.eligible\_clusters$
    else
      Provision cloud cluster
  else
    for γ in $\Delta.eligible\_clusters$ **do**
      if $\delta.replicas \times \delta.cpu\_req < \gamma.cpu.avail$ and $\delta.replicas \times \delta.memory\_req < \gamma.memory.avail$ **then**
        Append γ to $\Delta.selected\_clusters$
  if len($\Delta.selected\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.selected\_clusters$
    else
      Provision cloud cluster
switch $(placement - policy)$
case $traffic - aware$**:**
  Descending sort $\Delta.selected\_clusters$ by $\delta.ntk$
case $worst - fit$**:**
  Descending sort $\Delta.selected\_clusters$ by $\gamma.cpu.avail$ and $\gamma.memory.avail$
case $best - fit$**:**
  Ascending sort $\Delta.selected\_clusters$ by $\gamma.cpu.avail$ and $\gamma.memory.avail$
end switch
Place Δ on $\Delta.selected\_clusters$

controller updates the number of replicas of the MCD and the multi-cluster scheduler adjusts the number of replicas. If, on the other hand, the number of desired replicas is less than that of the current replicas, the controller waits for a configurable cool-down period (10 minutes by default) to avoid fluctuations before it updates the MCD. The details of the controller are as shown in Algorithm 3.

Algorithm 3: Multi-cluster horizontal pod autoscaling (MCHPA)

Input: Definition for MCHPA with target MCD Δ, θ, $\delta.min\_replicas$, and $\delta.max\_replicas$
Output: $\Delta.desired\_replicas$ **Parameters** : Cool-down period

while not exited **do**
  For the given MCD Δ, get MCD $\Delta.clusters$, $\delta.cpu\_req$
  for γ in $\Delta.clusters$ **do**
    Get $\delta.current\_replicas$, $\delta.cpu\_usage$
    Compute $\delta.desired\_replicas$ = ($\delta.current\_replicas \times \delta.cpu\_util$) / θ
    if $\delta.desired\_replicas < \delta.min\_replicas$ **then**
      $\delta.desired\_replicas \leftarrow \delta.min\_replicas$
    else if $\delta.desired\_replicas > \delta.max\_replicas$ **then**
      $\delta.desired\_replicas \leftarrow \delta.max\_replicas$
    if $\delta.desired\_replicas < \delta.current\_replicas$ **then**
      Wait for cool down period
    Append $\delta.desired\_replicas$ to $\Delta.desired\_replicas$
  return $\Delta.desired\_replicas$

3) Cloud Provisioner and Cluster Autoscaler: One of the objectives of mck8s is to dynamically provision *k8s* clusters from public cloud data centers to complement a multi-cluster environment when its clusters run out of resources. This avoids under-provisioning while avoiding the high cost of running clusters in the cloud even when they are underutilized.

The cloud provisioner and autoscaler manages the entire lifecycle of a *k8s* cluster in the cloud including provisioning, cluster autoscaling, and deprovisioning. The controller manages an object that is defined and deployed once with information about the cloud provider, region, and authentication credentials. The controller runs periodically and monitors the status of all MCDs deployed in the platform. When the controller finds one or more MCDs that could not be deployed because of a shortage of resources, it computes the number and size of nodes needed to host the MCDs. Then, it provisions a *k8s* cluster with the computed size and number of VMs using Cluster API[2] tool that allows creating *k8s* clusters declaratively from cloud providers. The cluster autoscaler adjusts the number of nodes of the *k8s* cluster in the cloud as the number of requested resources fluctuates. Finally, the controller removes the *k8s* cluster from the cloud altogether if it is under-utilized for a pre-defined amount of time.

4) Multi-Cluster Re-scheduler: The Multi-Cluster Re-scheduler manages the custom resource of the same name. The controller is deployed on the management cluster once and periodically checks the cloud cluster for MCDs deployed on it because of a shortage of resources on their preferred clusters. When the controller finds such MCDs, it passes these MCDs to the Multi-Cluster Scheduler so that it attempts to schedule them once again, in which case they will be deployed on their preferred clusters if enough resources are available. In so doing, the re-scheduler contributes to minimizing over-provisioning in the cloud cluster.

[2]https://github.com/kubernetes-sigs/cluster-api

Algorithm 4: Cloud cluster provisioner and autoscaler

Input: Definition for cloud cluster β, cloud provider information (region, credentials, etc.)
Output: β

while not exited **do**
  if $\exists\ \beta$ **then**
    if $\mid P^{i}_{pending} \mid > 0$ **then**
      Scale-out
      Calculate number and size of additional nodes
    else
      if number of nodes of cloud cluster == 1 **then**
        if number of deployment on cloud cluster == 0 **then**
          Remove cloud cluster
        else
          if $\exists$ nodes where sum of resource requests $<$ node allocatable **then**
            Scale-in
            Remove nodes
  else
    for Δ_i in $\{\Delta_1, \Delta_2, ..., \Delta_n\}$ **do**
      if $\Delta_i.status.message ==$ 'to_cloud' **then**
        total_cpu_req += $\delta_i.replicas \times \delta_i.cpu\_req$
        total_memory_req += $\delta_i.replicas \times \delta_i.memory\_req$
    Compute number and size of nodes for the cloud cluster
    Provision cloud cluster β
    return β

VI. Evaluation

We evaluate mck8s using four sets of experiments: (1) Scheduling of several short-running jobs and long-running services with a wide range of resource requests modeled after the Google cluster traces; (2) Autoscaling and placement policies as user traffic moves from one cluster to another; (3) Bursting of a multi-cluster deployment during autoscaling from a source cluster to other clusters and eventually to the cloud, along with service routing; and (4) Performance of mck8s in terms of deployment times for multi-cluster scheduling. We ran all experiments three times and the results from one of the runs is presented, except in the case of Experiment 2 where the results are the average of the three runs.

A. Experimental Setup

We perform our experiments in the Grid'5000 experimental testbed (see Figure 4). The management cluster that runs the KubeFed controllers is deployed in Rennes, and five *k8s* clusters are located in five different sites. Each cluster has one master node and five worker nodes. Clusters 1 and 5 use nodes with 4 CPU cores and 16 GB RAM, whereas clusters 2–4 use nodes with 2 CPU cores and 4 GB of RAM. Moreover, an OpenStack cluster in Nancy acts as the cloud platform where we provision a *k8s* cluster during cloud bursting.

We use Kubernetes v1.18.0, Kubernetes Federation v0.1.0-rc6, Cluster API v0.3.10 with OpenStack provider v0.3.1, Cilium v1.9.3, serf 0.8.2, and Prometheus Operator 0.45.0.

B. Multi-Cluster Scheduling

To evaluate the capability of mck8s' scheduler to place a variety of deployments and jobs, we deploy a workload based on the Google Cluster traces. The Google cluster traces contain information about thousands of containers with diverse resource requirements, duration, and inter-arrival rates [23]. In particular it exhibits heterogeneity in CPU and RAM request, inter-arrival rates, and job duration and has been extensively used to evaluate resource scheduling in the cloud [24].

We created a synthetic workload that matches the statistical distribution of the Google cluster trace, and augmented it with location information generated using a binomial distribution. Properties of the workload are shown in Figure 5 and Table III. We ran the workload for 60 minutes during which 1,126 tasks were created. We wait 30 more minutes to observe tasks finish running and free up resources. Tasks longer than 60 minutes are treated as long-running services (MCD) whereas shorter ones are treated as short-running jobs (MCJ). As new tasks are submitted to the management cluster, mck8s' scheduler goes through the filtering and prioritizing phases of scheduling and places a replica of the task on the preferred cluster as specified in the workload. In this experiment, substitution is enabled with substitutionPolicy 'nearest-first', meaning that if the preferred cluster is out of resources during placement, the scheduler places the task on the closest substitution cluster using network latency measured by Serf.

The stacked-area plot in Figure 6 shows the allocation of application pods on the member clusters of the multi-cluster environment. As MCDs and MCJs are submitted to the management cluster, the multi-cluster scheduler places the deployments and jobs preferably on their preferred clusters as specified in the manifest files if the latter have sufficient resources available. However, if a cluster does not have sufficient resources to place a deployment and since substitution is enabled, some deployments are placed on clusters that are close to the original preferred cluster instead. When all clusters

TABLE II: Inter-site network latency (RTT) in milliseconds in Grid'5000.

	Rennes	Nantes	Lille	Luxembourg	Nancy	Grenoble
Rennes	-	2.16	23.26	27.41	25.18	17.45
Nantes	2.16	-	22.21	26.29	24.16	16.38
Lille	23.26	22.21	-	11.88	9.70	12.06
Luxembourg	27.41	26.29	11.88	-	2.90	15.33
Nancy	25.18	24.16	9.70	2.90	-	13.14
Grenoble	17.45	16.38	12.06	15.33	13.14	-

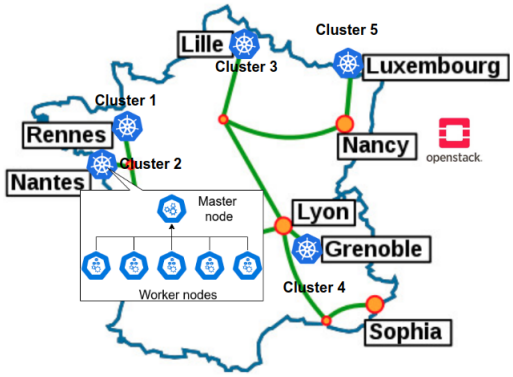
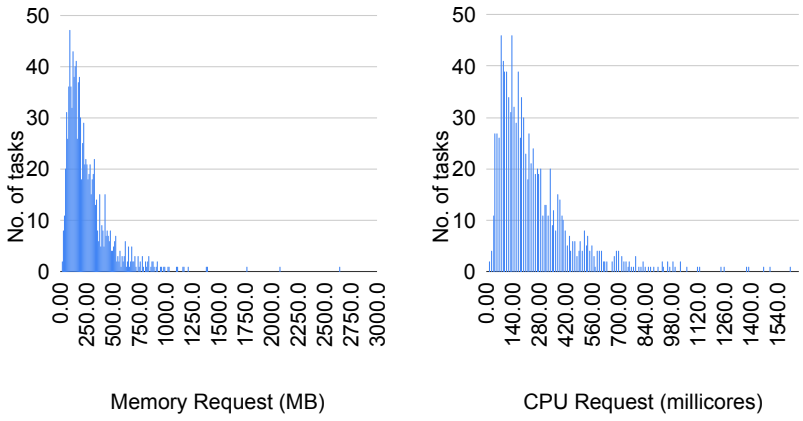


Fig. 4: Experimental setup in Grid'5000 consisting of a management cluster in Rennes, and five member clusters in Rennes, Nantes, Lille, Grenoble (France), and Luxembourg, and an OpenStack cluster in Nancy. Distances between sites range from 100 km to 850 km. Each *k8s* cluster has a master node and five worker nodes. Image adapted from the Grid'5000 website.



(a) Tasks memory request.

(b) Tasks CPU request.

Fig. 5: Characteristics of workload based on Google cluster traces.

TABLE III: Distribution of tasks across locations (clusters).

Location	Cluster 1 (Rennes)	Cluster 2 (Nantes)	Cluster 3 (Lille)	Cluster 4 (Grenoble)	Cluster 5 (Luxembourg)
No. of tasks	125	362	391	163	85

run out of resources at $t = 43\,min$, the cloud provisioner provisions a *k8s* cluster in the cloud where the multi-cluster scheduler places the deployments that could not be placed in the multi-cluster environment because of a shortage of resources. The broken lines show the total amount of CPU cores provisioned in the cloud, then scaled down and eventually decommissioned when no longer needed.

Figure 7 shows the per-cluster and overall CPU allocation, which is measured as the ratio of the total CPU request of pods to the total cluster CPU. We see that mck8s' substitution and policy results in a balanced CPU allocation per-cluster and overall, as opposed to that of KubeFed discussed in Section II. Again, in Figure 7 we see that mck8s' scheduling policy results in only a maximum of 6% of the submitted pods are pending as opposed to 65% in the case of KubeFed (Figure 1).

C. Autoscaling and Policy-Based Placement

In this experiment, we create a scenario that shows how an application deployed on one of the clusters responds to user traffic as the source of traffic moves. For this experiment, we deployed the Cilium cluster mesh on the clusters to enable cross-cluster service routing. The application used in this evaluation is a two-tier MCD consisting of an nginx front-end and a simple PHP web application as a backend. The frontend is deployed on all five clusters, whereas the backend is initially deployed on Cluster 2 with two replicas. We allocated 0.5 CPU cores[3] and 512 MB of memory to the pods of both the backend and frontend applications. To enable autoscaling, the corresponding MCHPA object is also applied for the backend, with CPU as the metric and 75% CPU utilization threshold. Then, constant user traffic with 10 concurrent users is applied to the application for 7 minutes from one source of traffic starting at Cluster 3 and then consecutively moving to Clusters 4, 5, 1, and 2. Three scheduling policies (i.e., traffic-aware, worst-fit, and best-fit) are evaluated.

For the sake of space, we only show the results from traffic-aware placement in Figure 8. We see that initially the backend was deployed on Cluster 2 and even though the source of traffic has moved to Cluster 3 it still receives the traffic thanks to the load balancing by Cilium. When the scheduler tries to select the appropriate cluster for the application following the update from the autoscaler during the next cycle, it selects Cluster 3 as the frontend on this cluster is receiving the most traffic. As a result, the backend is moved to Cluster 3. Moreover, the number of replicas is increased to accommodate the traffic from 10 concurrent users. In the same manner, the backend follows the user traffic to clusters 4, 5, 1, and 2. By doing so, mck8s makes sure that the application is deployed closer to end-users and the number of replicas is adjusted to make sure that the application meets its performance requirements.

D. Multi-Cluster Horizontal Pod Autoscaling and Bursting

We now evaluate the autoscaling, bursting, and cloud provisioning features of mck8s working together that allow an MCD faced with a sudden increase in user traffic to make use of neighboring clusters as well as resources in the cloud. The results are shown in Figures 9–10.

Similar to the previous experiment, we deployed Cilium cluster mesh on the clusters to enable inter-cluster service routing and used the same two-tier application for evaluation. We allocated 0.5 CPU and 512 MB of memory to the pods of both the frontend application, whereas 1 core of CPU and 1024 MB memory were allocated to the backend. Initially, we deploy five replicas of the front end and two replicas of the

[3]In k8s, one CPU is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. A Container with request of 0.5 CPU cores is guaranteed half as much CPU as one that asks for 1 CPU core. Read more here. https://bit.ly/3sueJ7E

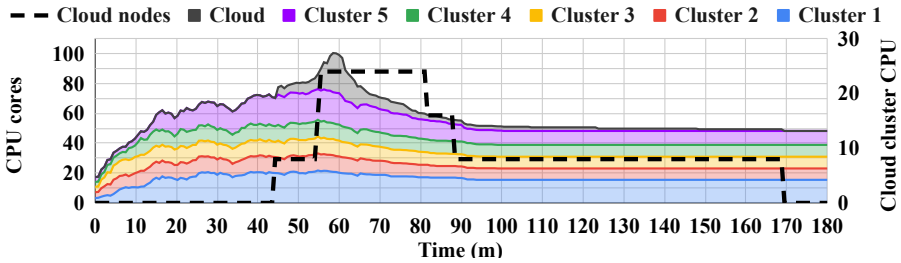


Fig. 6: Multi-cluster scheduling pods CPU request and cloud cluster lifecycle. Dashed line represents CPU cores of cloud nodes, whereas the stacked area represents the total CPU request of the pods running in the clusters.

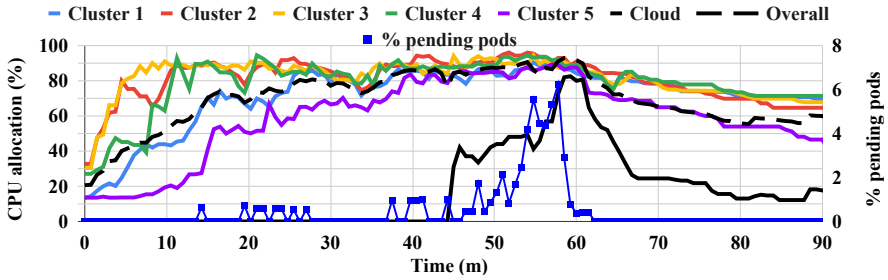


Fig. 7: Multi-cluster scheduling: Per-cluster and overall CPU allocation and percentage of pending pods.

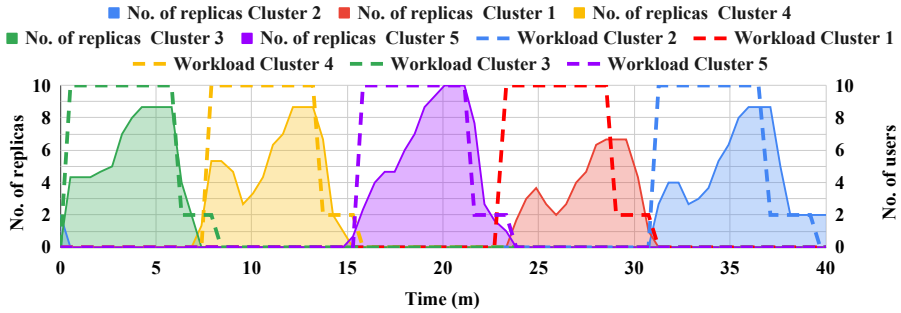


Fig. 8: Multi-cluster horizontal pod autoscaling and traffic aware placement. Dashed lines represent no. of users, whereas solid areas represent no. of replicas.

backend on Cluster 2. An MCHPA instance is created for the backend, with 2 minimum number of replicas, 100 maximum number of replicas, CPU as the scaling metric, and 50% CPU utilization threshold. MCHPA is configured with a cool-down period of 10 minutes, whereas the cluster autoscaler is configured with a scale-down delay and deprovisioning delay of 10 minutes and 20 minutes, respectively.

The workload used for this evaluation and shown using broken lines in Figure 9 is based on the San Francisco taxi traces [25] and it is applied to the frontend service on Cluster 2. The autoscaler adjusts the number of replicas in response to the changing workload and the scheduler configured with the traffic-aware placement policy, bursting enabled and the nearest-first bursting policy places the replicas in the right clusters. As can be seen in Figures 9 the application bursts from Cluster 2 successively to clusters 1, 4, 3, 5. When Cluster 5 runs out of resources the cloud provisioner provisions a *k8s* cluster on the OpenStack cloud and joins it to the multi-cluster federation so that more replicas are deployed on it. As the user traffic decreases after 60 minutes, the application is "pulled back" to the original Cluster 2. Figure 10 shows the full lifecycle of the cloud cluster as it is created, autoscaled, and finally deprovisioned. After the workload starts decreasing at around 47 minutes, we notice a slow scale down of replicas as well as a scale-down and deprovisioning of the cloud cluster due to the cool-down periods and the relatively low CPU utilization threshold.

E. Deployment times

To demonstrate the performance of scheduling in mck8s we perform a few measurements of deployment time by varying the number of replicas of an MCD from 1 to 100 and the placement policies (cluster-affinity and traffic-aware). In this experiment, we allocate 0.5 CPU cores and 512 MB of memory to the pods of the MCD. The results are shown in Table IV. We see that, in general, deployment time increases as the number of replicas to be deployed increases because the number of clusters that the scheduler has to filter and prioritize increases as well, especially if the resource request of the deployment is relatively high or the clusters have relatively smaller capacity or available resources. In the case of the traffic-aware placement policy, the deployment times are

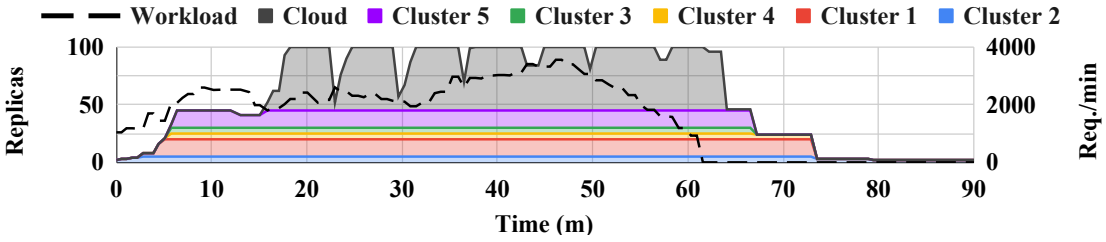


Fig. 9: No. replicas of the MCD during multi-cluster horizontal pod autoscaling and cloud bursting. Dashed lines represent requests per minute generated by the workload.

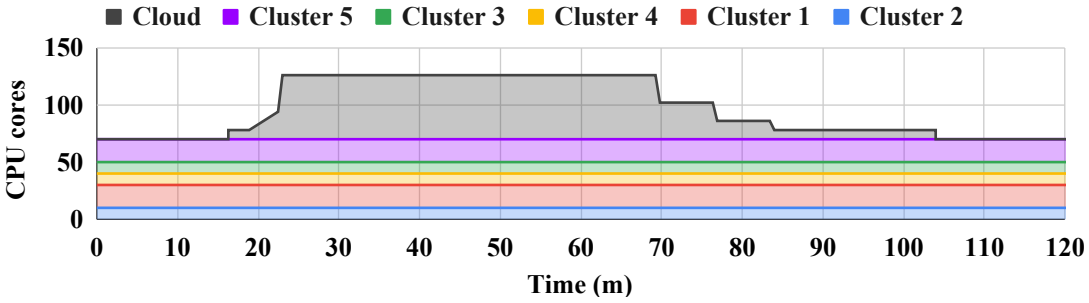


Fig. 10: Total CPU cores provided by clusters including cloud cluster's full life cycle during multi-cluster horizontal pod autoscaling and bursting.

TABLE IV: Multi-cluster scheduling deployment times as no. of replicas change.

No. of replicas	Deployment time (seconds)	
	Traffic aware with bursting	Cluster Affinity with bursting
1	8.63	1.88
10	8.67	1.78
100	9.71	11.53

higher overall than those of the cluster-affinity policy, because the scheduler has to sort all clusters based on the received traffic before selecting the appropriate clusters whereas in the cluster-affinity placement policy the scheduler already knows the preferred cluster to be selected if it has enough resources available, otherwise, the scheduler looks for a substitute which is common as the number of replicas grows. We see that the deployment time varies between 1.88s for 1 replica to 11.52 s for 100 replicas for the cluster-affinity placement policy. On the other hand, it takes 8.62 s for deploying 1 replica and 9.71 s to place 100 replicas with the traffic-aware placement policy. This illustrates that the mck8s scheduler scales well as the number of replicas increases. And since the scheduler is the core component of mck8s, this shows that mck8s is well suited to serve as an orchestration platform in a geo-distributed multi-cluster environment that is expected to handle thousands of applications within a short amount of time.

VII. Conclusions

In this work, we address the gap in integrated resource management of geo-distributed clusters. We propose mck8s, a generic and integrated orchestration platform for multi-cloud deployments with policy-based scheduling, autoscaling, and cloud bursting capabilities. Although mck8s introduces new controllers and interfaces, we argue that it can be easily adopted because of its simple integration with vanilla Kubernetes. Using realistic experiments, we show that mck8s balances the resource allocation across multiple geo-distributed clusters and reduces the fraction of pending pods from 65% in the case of Kubernetes Federation to 6% for the same workload. The mck8s implementation is freely available under a liberal open-source license[4].

In addition to the simple heuristics presented in this paper, mck8s may be further extended to include more sophisticated and proactive placement and autoscaling algorithms to address different multi-cluster use cases.

References

[1] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. R. Souza Jr., M. A. Tamiru, and L. Wu, "Fog computing applications: Taxonomy and requirements," *CoRR*, vol. abs/1907.11621, 2019. [Online]. Available: http://arxiv.org/abs/1907.11621.

[2] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-latency-aware fog application replicas autoscaler," in *Proc. MASCOTS*, 2020.

[3] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Computing Surveys*, vol. 47, no. 1, 2014.

[4] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, 2009.

[4]mck8s – https://github.com/moule3053/mck8s

[5] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, 2012.

[6] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti, "Cloud federations in Contrail," in *Proc. Euro-Par*, 2011.

[7] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future generation computer systems*, vol. 28, no. 2, 2012.

[8] T. Guo, U. Sharma, T. Wood, S. Sahu, and P. Shenoy, "Seagull: Intelligent cloud bursting for enterprise applications," in *Proc. USENIX ATC*, 2012.

[9] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan, "Towards secure cloud bursting, brokerage and aggregation," in *Proc. ECOWS*, 2010.

[10] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Transactions on Internet Technology*, vol. 13, no. 3, 2014.

[11] M. R. H. Farahabady, Y. C. Lee, and A. Y. Zomaya, "Pareto-optimal cloud bursting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, 2013.

[12] Y. C. Lee and B. Lian, "Cloud bursting scheduler for cost efficiency," in *Proc. IEEE CLOUD*, 2017.

[13] G. L. Stavrinides and H. D. Karatza, "Cost-aware cloud bursting in a fog-cloud environment with real-time workflow applications," *Concurrency and Computation: Practice and Experience*, 2020.

[14] F. Faticanti, J. Zormpas, S. Drozdov, K. Rausch, O. A. García, F. Sardis, S. Cretti, M. Amiribesheli, and D. Siracusa, "Distributed cloud intelligence: Implementing an ETSI MANO-compliant predictive cloud bursting solution using openstack and kubernetes," in *Intl. Conf. on the Economics of Grids, Clouds, Systems, and Services*, 2020.

[15] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. MCC workshop on Mobile cloud computing*, 2012.

[16] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Proc. IEEE HotWeb*, 2015.

[17] A. J. Fahs and G. Pierre, "Tail-latency-aware fog application replica placement," in *Proc. ICSOC*, 2020.

[18] D. Kim, H. Muhammad, E. Kim, S. Helal, and C. Lee, "TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform," *Applied Sciences*, vol. 9, no. 1, 2019.

[19] F. Rossi, V. Cardellini, and F. Lo Presti, "Elastic deployment of software containers in geo-distributed computing environments," in *Proc. IEEE ISCC*, 2019.

[20] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, no. 1, 2020.

[21] M. Nardelli, V. Cardellini, and E. Casalicchio, "Multi-level elastic deployment of containerized applications in geo-distributed environments," in *Proc. IEEE FiCloud*, 2018.

[22] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, 2003.

[23] C. Reiss, J. Wilkes, and J. Hellerstein, *Google cluster-usage traces: Format+ schema*, Google White Paper, 2011.

[24] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM SOCC*, 2012.

[25] M. A. Hoque, X. Hong, and B. Dixon, "Analysis of mobility patterns for urban taxi cabs," in *Proc. IEEE ICNC*, 2012.